1.Write a program to implement **k-Nearest Neighbour algorithm** to classify the iris data set. Print both correct and wrong predictions.

import numpy as np

import pandas as pd

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn import metrics

names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']

dataset = pd.read\_csv("8-dataset.csv", names=names)

X = dataset.iloc[:, :-1]

y = dataset.iloc[:, -1]

print(X.head())

Xtrain, Xtest, ytrain, ytest = train\_test\_split(X, y, test\_size=0.10)

classifier = KNeighborsClassifier(n\_neighbors=5).fit(Xtrain, ytrain)

ypred = classifier.predict(Xtest)

i = 0

print ("\n-------------------------------------------------------------------------")

print ('%-25s %-25s %-25s' % ('Original Label', 'Predicted Label', 'Correct/Wrong'))

print ("-------------------------------------------------------------------------")

for label in ytest:

print ('%-25s %-25s' % (label, ypred[i]), end="")

if (label == ypred[i]):

print (' %-25s' % ('Correct'))

else:

print (' %-25s' % ('Wrong'))

i = i + 1

print ("-------------------------------------------------------------------------")

print("\nConfusion Matrix:\n",metrics.confusion\_matrix(ytest, ypred))

print ("-------------------------------------------------------------------------")

print("\nClassification Report:\n",metrics.classification\_report(ytest, ypred))

print ("-------------------------------------------------------\------------------")

print('Accuracy of the classifer is %0.2f' % metrics.accuracy\_score(ytest,ypred))

print ("-------------------------------------------------------------------------")

**Program 2**

**Develop a program to apply K-means algorithm to cluster a set of data stored in .CSV file. Use the same data set for clustering using EM algorithm. Compare the results of these two algorithms and comment on the quality of clustering.**

from sklearn.cluster import KMeans

from sklearn import preprocessing

from sklearn.mixture import GaussianMixture

from sklearn.datasets import load\_iris

import sklearn.metrics as sm

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

dataset=load\_iris()

X=pd.DataFrame(dataset.data)

X.columns=['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y=pd.DataFrame(dataset.target)

y.columns=['Targets']

plt.figure(figsize=(14,7))

colormap=np.array(['red','lime','black'])

# REAL PLOT

plt.subplot(1,3,1)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y.Targets],s=40)

plt.title('Real')

# K-PLOT

plt.subplot(1,3,2)

model=KMeans(n\_clusters=3)

model.fit(X)

predY=np.choose(model.labels\_,[0,1,2]).astype(np.int64)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[predY],s=40)

plt.title('KMeans')

# GMM PLOT

scaler=preprocessing.StandardScaler()

scaler.fit(X)

xsa=scaler.transform(X)

xs=pd.DataFrame(xsa,columns=X.columns)

gmm=GaussianMixture(n\_components=3)

gmm.fit(xs)

y\_cluster\_gmm=gmm.predict(xs)

plt.subplot(1,3,3)

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y\_cluster\_gmm],s=40)

plt.title('GMM Classification')

**PROGRAM 3**

**Implement the non-parametric Locally Weighted Regressionalgorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.**

import numpy as np

import matplotlib.pyplot as plt

X=np.linspace(-3,3,1000)

print(X)

X+=np.random.normal(scale=0.05,size=1000)

Y=np.log(np.abs((X\*\*2)-1)+0.5)

print(Y)

np.linspace(2.0, 3.0, num=5)

plt.scatter(X,Y,alpha=0.32)

def local\_regression(x0,X,Y,tau):

x0=np.r\_[1,x0]

X=np.c\_[np.ones(len(X)),X]

xw=X.T \*radial\_kernel(x0,X,tau)

print(xw)

beta=np.linalg.pinv(xw@X)@xw@Y

return x0@beta

**Exp. No. 4. Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets**.

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)

y = np.array(([92], [86], [89]), dtype=float)

X = X/np.amax(X,axis=0)

y = y/100

def sigmoid (x):

return 1/(1 + np.exp(-x))

def derivatives\_sigmoid(x):

return x \* (1 - x)

epoch=5000

lr=0.1

inputlayer\_neurons = 2

hiddenlayer\_neurons = 3

output\_neurons = 1

wh=np.random.uniform(size=(inputlayer\_neurons,hiddenlayer\_neurons))

bh=np.random.uniform(size=(1,hiddenlayer\_neurons))

wout=np.random.uniform(size=(hiddenlayer\_neurons,output\_neurons))

bout=np.random.uniform(size=(1,output\_neurons))

for i in range(epoch):

hinp1=np.dot(X,wh)

hinp=hinp1 + bh

hlayer\_act = sigmoid(hinp)

outinp1=np.dot(hlayer\_act,wout)

outinp= outinp1+ bout

output = sigmoid(outinp)

EO = y-output

outgrad = derivatives\_sigmoid(output)

d\_output = EO\* outgrad

EH = d\_output.dot(wout.T)

hiddengrad = derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer = EH \* hiddengrad

wout += hlayer\_act.T.dot(d\_output) \*lr

wh += X.T.dot(d\_hiddenlayer) \*lr

print("Input: \n" + str(X))

print("Actual Output: \n" + str(y))

print("Predicted Output: \n" ,output)

**EXP NO 5:**

**Demonstrate Genetic algorithm by taking a suitable data for any simple application.**

import random

import string

target = "Hello, World!"

population\_size = 1000

generations = 1000

def fitness(individual):

return sum([1 for i, j in zip(individual, target) if i == j])

population = [''.join(random.choices(string.ascii\_letters + ' ', k=len(target))) for \_ in range(population\_size)]

for generation in range(generations):

scores = [fitness(individual) for individual in population]

parents = random.choices(population, weights=scores, k=population\_size)

population = [parent[:len(target)//2] + random.choice(parents)[len(target)//2:] for parent in parents]

population = [''.join([char if random.random() > 0.01 else random.choice(string.ascii\_letters + ' ') for char in individual]) for individual in population]

print(max(population, key=fitness))

print("Final best individual: ", max(population, key=fitness))

**EXP NO 6**

**Demonstrate Q learning algorithm with suitable assumption for a problem statement.**

import numpy as np

Q = np.zeros([5, 5])

gamma = 0.8

alpha = 0.9

R = np.array([[0, 0, 0, 0, 100],

[0, 0, 0, 100, -100],

[0, 0, 0, 100, -100],

[0, 100, 100, 100, 100],

[100, -100, -100, 100, 100]])

for episode in range(100):

state = np.random.randint(0, 5)

while state != 4:

action = np.random.randint(0, 5)

Q[state, action] = R[state, action] + gamma \* max(Q[action, :])

state = action

Q /= np.max(Q)

print(Q)